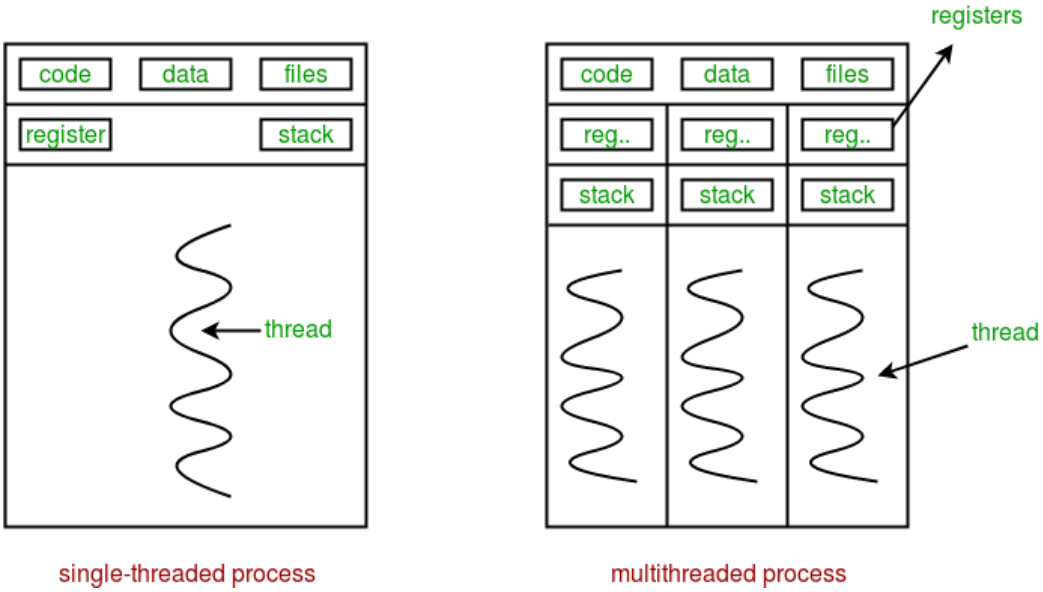
Productor Consumidor



1. **Introducción**
2. **Versión 1**
3. **Versión 2**
4. **Versión 3**
5. **Versión 4**
6. **Fuentes de información**

1. Introducción

El problema productor-consumidor es un ejemplo clásico donde es necesario dar un tratamiento independiente a un conjunto de datos que se generan de forma más o menos aleatoria o al menos de forma que no es posible predecir en qué momento. se generará un dato. Para evitar el uso excesivo de los recursos informáticos a la espera de la llegada de los datos, el sistema prevé dos tipos de procesos: los productores, encargados de obtener los datos a tratar, y los consumidores, especializados en el tratamiento de los datos obtenidos por los productores.

Mira el ejemplo resuelto del problema de **productores-consumidores**

2. Versión 1

Vamos a desarrollar el problema **Productor-Consumidor**, primero veamos cuál es el problema: crearemos dos tipos de hilos: un `Productor` que pondrá algunos datos (por ejemplo, un número entero) en un objeto (lo llamaremos `SharedData`), y un `Consumidor` que obtendrá estos datos. Nuestra clase `SharedData.java` es esta:

```
1  public class SharedData {
2
3      int data;
4
5      public int get() {
6          return data;
7      }
8
9      public void put(int newData) {
10         data = newData;
11     }
12 }
```

Y las clases `Producer.java`:

```
1  public class Producer extends Thread {
2
3      SharedData data;
4
5      public Producer(SharedData data) {
6          this.data = data;
7      }
8
9      @Override
10     public void run() {
11         for (int i = 0; i < 50; i++) {
12             data.put(i);
13             System.out.println("Produced number " + i);
14             try {
15                 Thread.sleep(10);
16             } catch (Exception e) {
17             }
18         }
19     }
20 }
```

y `Consumer.java`:

```
1  public class Consumer extends Thread {
2
3      SharedData data;
4
```

```
        this.data = data;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            int n = data.get();
            System.out.println("Consumed number " + n);
            try {
                Thread.sleep(10);
            } catch (Exception e) {
            }
        }
    }
}
```

La aplicación `Test.java` principal creará un objeto `SharedData` y un subproceso de cada tipo, e iniciará ambos.

```
public class Test {
    public static void main(String[] args) {
        SharedData sd = new SharedData();
        Producer p = new Producer(sd);
        Consumer c = new Consumer(sd);
        p.start();
        c.start();
    }
}
```

En el resultado podemos observar algunos problemas:

```
run:
Consumed number 0
Produced number 0
Consumed number 0
Produced number 1
Consumed number 1
Produced number 2
Produced number 3
Consumed number 3
Produced number 4
```

3. Versión 2

Podríamos pensar que si simplemente agregamos la palabra clave `synchronized` a los métodos `get` y `put` de la clase `SharedData`, resolveríamos el problema:

```
public class SharedData {
    int data;
    public synchronized int get() {
        return data;
    }
    public synchronized void put(int newData) {
        data = newData;
    }
}
```

Sin embargo, si volvemos a ejecutar el programa, podemos notar que sigue fallando:

```
run:
Consumed number 0
Produced number 0
Consumed number 0
Produced number 1
Produced number 2
Consumed number 1
Produced number 3
Consumed number 3
Produced number 4
```

4. Versión 3

De hecho, hay dos problemas que tenemos que resolver. Pero empecemos por lo más importante: el productor y el consumidor tienen que trabajar coordinadamente: en cuanto el productor pone un número, el consumidor puede conseguirlo, y el productor no puede producir más números hasta que el consumidor consigue los anteriores.

Para hacer esto, necesitamos agregar algunos cambios a nuestra clase `SharedData`. En primer lugar, necesitamos una bandera que les diga a los productores y consumidores quién es el siguiente. Dependerá de si hay nuevos datos para consumir (turno del consumidor) o no (turno del productor).

```
1  public class SharedData {
2
3      int data;
4      boolean available = false;
5
6      public synchronized int get() {
7          available = false;
8          return data;
9      }
10
11     public synchronized void put(int newData) {
12         data = newData;
13         available = true;
14     }
15 }
```

5. Versión 4

Además, debemos asegurarnos de que los métodos `get` y `put` se llamen alternativamente. Para hacer esto, necesitamos usar la bandera booleana y los métodos `wait` y `notify` / `notifyAll`, así:

```
public class SharedData {

    int data;
    boolean available = false;

    public synchronized int get() {
        if (!available) {
            try {
                wait();
            } catch (Exception e) {
            }
        }
        available = false;
        notify();
        return data;
    }

    public synchronized void put(int newData) {
        if (available) {
            try {
                wait();
            } catch (Exception e) {
            }
        }
        data = newData;
        available = true;
        notify();
    }
}
```

Observe cómo usamos los métodos `esperar` y `notificar`. Con respecto al método `get` (llamado por el `Consumer`), si no hay nada disponible, esperamos. Luego obtenemos el número, establecemos el indicador en falso nuevamente y notificamos al otro hilo.

En el método `put` (llamado por el `Producer`), si hay algo disponible, esperamos hasta que alguien nos notifique. Luego `configuramos` los nuevos datos, configuramos el indicador en `verdadero` nuevamente y notificamos al otro hilo.

Si ambos subprocesos intentan llegar a la sección crítica al mismo tiempo, el `Consumidor` tendrá que esperar (el indicador se establece en `falso` al principio), y el `Productor` establecerá los primeros datos que se consumirán. A partir de ese momento, los hilos se alternarán en la sección crítica, consumiendo y produciendo nuevos datos cada vez.

6. Fuentes de información

- Wikipedia
- Programación de servicios y procesos - FERNANDO PANIAGUA MARTÍN [Paraninfo]
- Programación de Servicios y Procesos - ALBERTO SÁNCHEZ CAMPOS [Ra-ma]
- Programación de Servicios y Procesos - Mª JESÚS RAMOS MARTÍN - [Garceta] (1ª y 2ª Edición)
- Programación de servicios y procesos - CARLOS ALBERTO CORTIJO BON [Sintesis]
- Programació de serveis i processos - JOAR ARNEDO MORENO, JOSEP CAÑELLAS BORNAS i JOSÉ ANTONIO LEO MEGÍAS [IOC]
- GitHub repositories:
    - https://github.com/ajcpro/psp
    - https://oscarmaestre.github.io/servicios/index.html
    - https://github.com/juanro49/DAM/tree/master/DAM2/PSP
    - https://github.com/pablohs1986/dam_psp2021
    - https://github.com/Perju/DAM
    - https://github.com/eldiegoch/DAM
    - https://github.com/eldiegoch/2dam-psp-public
    - https://github.com/franlu/DAM-PSP
    - https://github.com/ProgProcesosYServicios
    - https://github.com/joseluisgs
    - https://github.com/oscarnovillo/dam2_2122
    - https://github.com/PacoPortillo/DAM_PSP_Tarea02_La-Cena-de-los-Filosofos